FVPP: Java PathFinder

Primjer provjere modela za programe pisane u Javi

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Sadržaj

- O Java PathFinderu
- Struktura Java PathFindera
- Specifikacija svojstava programa
- Napredne teme i proširenja
- O 2. domaćoj zadaći

O JAVA PATHFINDERU

P. Mehlitz

N. Rungta

C. Pasareanu

W. Visser

Što je Java PathFinder?

- Java PathFinder (dalje: JPF) je radni okvir i skup alata za formalnu verifikaciju programa pisanih u programskom jeziku Java i to metodom provjere modela
- Vrlo složen radni okvir koji nudi niz mogućnosti proširenja
- Jedan od rijetkih uspješnih alata za izravnu verifikaciju izvornog koda programa (ne apstrakcije programa)
- Razvio ga je tim znanstvenika u NASA-i u suradnji s većim brojem sveučilišta, pojedinaca i poduzeća u svijetu (više desetaka takvih subjekata, od poduzeća se navodi Fujitsu)
- Alat otvorenog koda koji je distribuiran pod Apache 2.0 licencom
- Jezgra JPF-a je **virtualni stroj (VM) koji se izvodi iznad Javinog virtualnog stroja** kako bi omogućio provjeru modela korisničkih programa

Razvoj JPF-a

1999. - projekt je započeo razvojem JPF-a kao translatora podskupa Jave 1.0 u jezik PROMELA za alat SPIN koji se koristi za formalnu verifikaciju

2000. - reimplementacija sustava kao virtualnog stroja za provjeru modela (tipično otkrivanje kvarova pri istovremenom izvođenju)

2003. - uvođenje sučelja za proširenja

2005. - sustav postaje otvorenog koda na sjedištu SourceForge (http://javapathfinder.sourceforge.net/)

2008.-danas - poboljšanja koda sudjelovanjem na Google Summer of Code

https://summerofcode.withgoogle.com/programs/2022/organizations/the-jpf-team

2009. - sustav je prebačen na vlastiti poslužitelj koji podržava proširenja i Wiki stranice (http://babelfish.arc.nasa.gov/trac/jpf)

2018. - dio sustava je prebačen na Github te ga održava zajednica:

https://github.com/javapathfinder

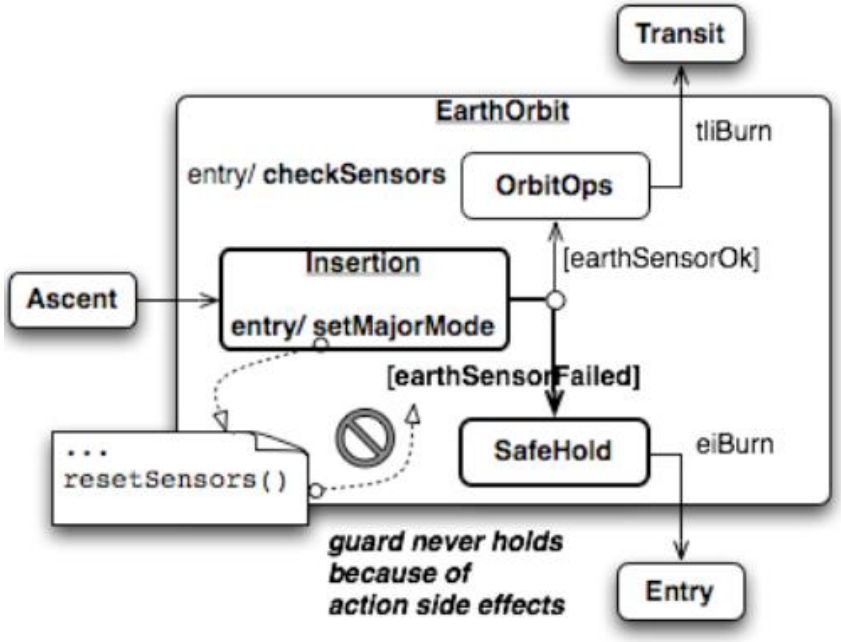
https://github.com/javapathfinder/jpf-core/wiki

2019. - poslužitelj na babelfish.arc.nasa.gov zatvoren za javnost

Zašto je razvijen JPF?

- **Pokrivenost linija koda** je nužno provjeriti kod kritičnih aplikacija – npr. lansiranja orbitera (NASA)
- Automatsko kodiranje UML-dijagrama stanja u program u Javi i isprobavanje može li se doći do svih dijelova koda – **analiza dosezljivosti**

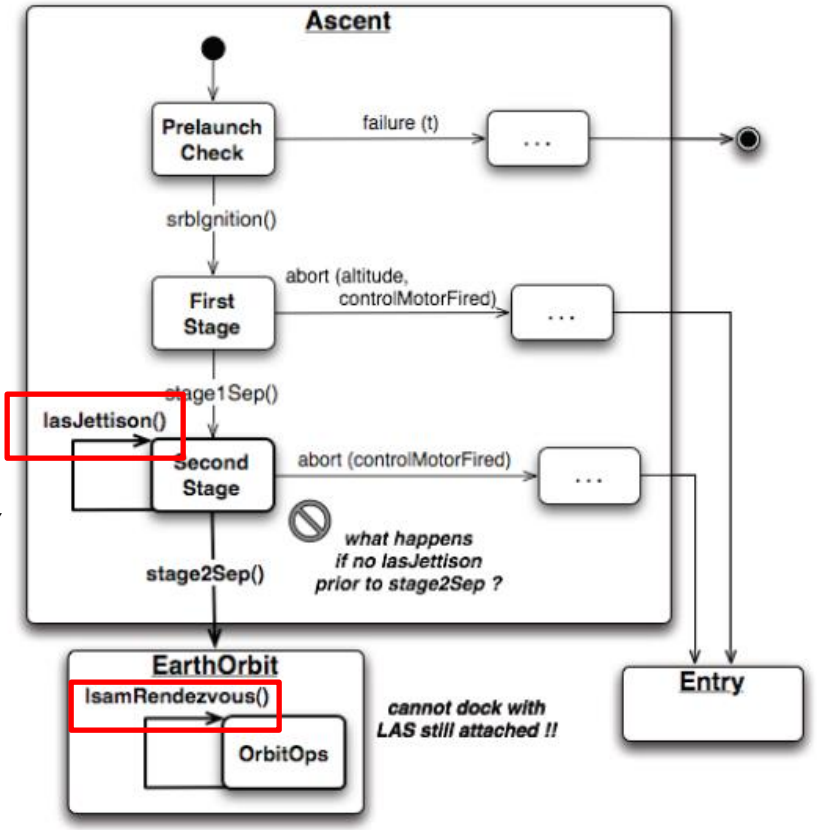
- U ovom primjeru, akcija `checkSensors()` isprobava senzore prilikom ulaska u stanje `EarthOrbit`, no varijabla vezana uz senzore (`earthSensorOk/Failed`) se provjerava tek kasnije – moguće je da senzore akcija `setMajorMode()` resetira tako da stanje `SafeHold` postane **nedosezljivo**



Zašto je razvijen JPF?

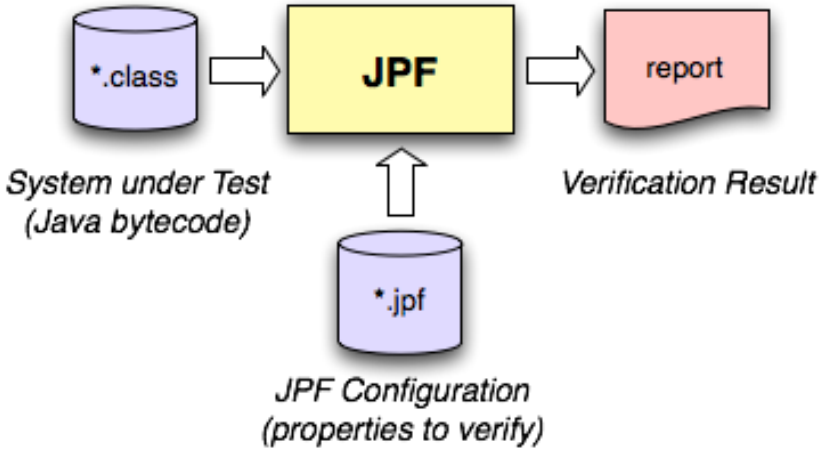
- Događaj `lasJetisson()` – odvajanje *launch abort* sustava (LAS) s letjelice
- `lsamRendevous()` – spajanje s drugom letjelicom – ono neće uspjeti ako se ne izvede `lasJetisson()` u stanju *Second Stage*, no to se ne osigurava u dijagramu
- Ascent i EarthOrbit module su možda čak radili različiti inženjeri
- Uvode se **tvrdnje** koje trebaju biti zadovoljene:

```
class OrbitOps {…
  void lsamRendezvous(){…
  assert !spacecraft.contains(LAS) :
  ”lsamRendezvous with LAS attached”
  …
 } …
}
```



Što radi Java PathFinder?

- JPF radi provjeru modela nad **kompajliranim Javinim aplikacijama** – aplikacija koja se provjerava naziva se **"*System under Test*" (SUT)**
- Java aplikacije mogu biti pisane na tri načina:
  1. **neovisno o JPF-u** (najčešći slučaj)
  2. **s podrškom za JPF** – sadržavaju određene anotacije koje koristi JPF
  3. **ovisno o JPF-u** – pisane specifično za JPF (rjeđi slučaj)
- Specifikacija svojstava koja SUT treba zadovoljiti radi se kroz konfiguraciju JPF-a (skup konfiguracijskih datoteka)
- JPF generira izvještaj (engl. *report*) u kojem detaljno piše rezultat provjere modela



Kada koristiti JPF?

- JPF nije pogodan za korištenje nad sekvencijskim programima s malim brojem dobro definiranih ulaznih vrijednosti – ***unit-testovi* su prikladniji za te primjene!**
- JPF se koristi za:

  1. Istraživanje alternativnih izvršavanja
     - **Sekvence raspoređivanja – konkurentne aplikacije** – najčešći razlog korištenja
     - **Varijacije u ulaznim podacima**
     - **Reakcije na događaje okoline**
     - **Izbor kontrolnog toka programa** – sustavno istraživanje strukture programa
  2. Inspekciju izvršavanja programa
     - Implementacija analizatora prekrivanja koda ili specifičnih neinvazivnih testova koji mogu otkriti uvjete koje je inače teško otkriti (npr. preljev, podljev)

Što otkriva JPF?

- **Jezgra JPF-a** (`jpf-core` projekt) otkriva takozvana **"svojstva nefunkcioniranja"** (engl. *non-functional properties*)
  - **Svojstva koja se ne bi smjela ispoljiti ni u jednoj aplikaciji, a ona uključuju: potpuni zastoj, utrke, neuhvaćene iznimke, provjere tvrdnji (`assert`).**
- **Proširenja JPF-a** otkrivaju razna korisnički specificirana svojstva – uglavnom korištenjem **"slušača" (engl. *listener*)**
  - Dodaci koji omogućuju da se pozorno prate sve akcije koje radi JPF, npr. izvođenje određenih instrukcija, stvaranje objekata, dolaska do određenog stanja programa, itd.
- JPF omogućuje prikaz **kompletne povijesti izvršavanja – traga** (engl. *trace*), čak do razine svake instrukcije *bytecode*, koja je dovela do otkrivanja pogreške.

Primjer otkrivanja utrke

```
public class Racer implements Runnable {
     int d = 42;
     public void run () {
          doSomething(1001);
          d = 0;                              // (1)
     }

   public static void main (String[] args){
          Racer racer = new Racer();
          Thread t = new Thread(racer);
          t.start();
          doSomething(1000);
          int c = 420 / racer.d;              // (2)
          System.out.println(c);
     }

     static void doSomething (int n) {
          // not very interesting..
          try { Thread.sleep(n); }
          catch (InterruptedException ix) {}
     }
}
```

Dvije dretve pristupaju istoj varijabli `d`. Dijeljenje s nulom se događa ako druga dretva (ona stvorena iz izvorne unutar metode `main`) izvede naredbu (1) prije nego što prva dretva izvede naredbu (2). To je moguće zbog načina kako se dretve raspoređuju za izvođenje.

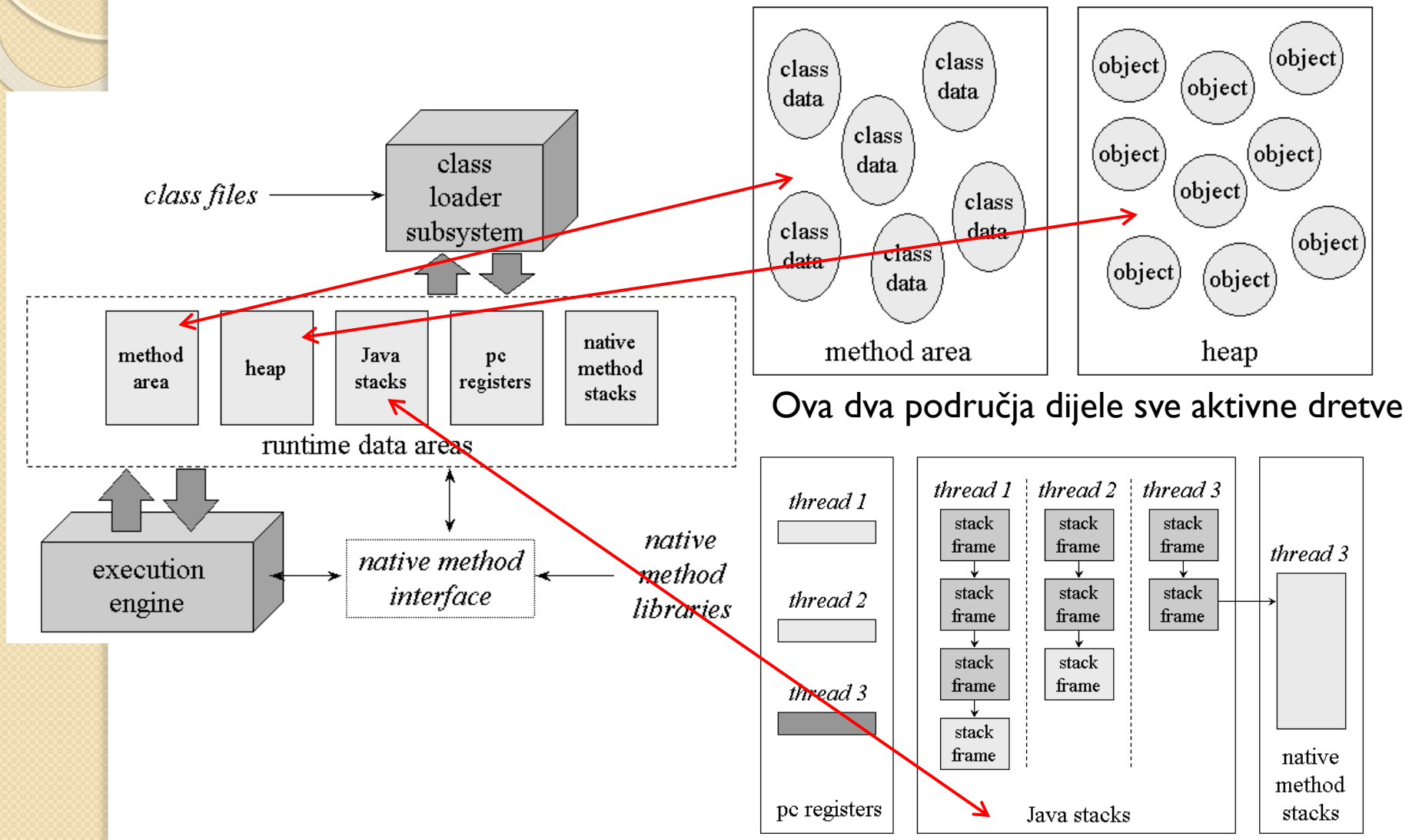
Bez ikakvog dodatnog podešavanja, JPF će pronaći neuhvaćenu iznimku `java.lang.ArithmeticException – division by zero`

Uz podešavanje (korištenje slušača `PreciseRaceDetector`), navest će točno retke koda (1) i (2) koji uzrokuju utrku dretvi.

Kako radi Java PathFinder?

- JPF je virtualni stroj koji izvršava naš SUT ne samo jednom (kao uobičajeni Javin VM), već na sve moguće načine
- Ako nađe mjesto kvara, onda izvještava o svakom koraku od početka do mjesta kvara
- U uobičajenom načinu izvršavanja (`jpf-core`), JPF je alat za provjeru modela s **eksplicitnim pamćenjem stanja izvođenja programa** (engl. *explicit state model checker*). To znači da pamti:
  - **okvire stoga dretvi (engl. *stack frames*)**
  - **objekte na gomili (engl. *heap objects*) i**
  - **stanja dretvi (engl. *thread states*)**
- Eksplicitno pamćenje stanja dovodi do **problema eksplozije broja stanja** (engl. *state explosion problem*)

Unutarnji pogled na ustroj Javinog virtualnog stroja (JVM)



Ova dva područja dijele sve aktivne dretve

Unutarnji pogled na ustroj Javinog virtualnog stroja (JVM)

- ***Class files*** – prevedeni kod – međukod (*bytecode*) razreda aplikacije i dodatne informacije
- ***Class loader subsystem*** – mehanizam za učitavanje tipova (razreda i sučelja) zadanih s **punim kvalificirajućim imenom** (npr. `java.awt.Rectangle`)
- ***Execution engine*** – mehanizam koji izvršava instrukcije međukoda

Unutarnji pogled na ustroj Javinog virtualnog stroja (JVM)

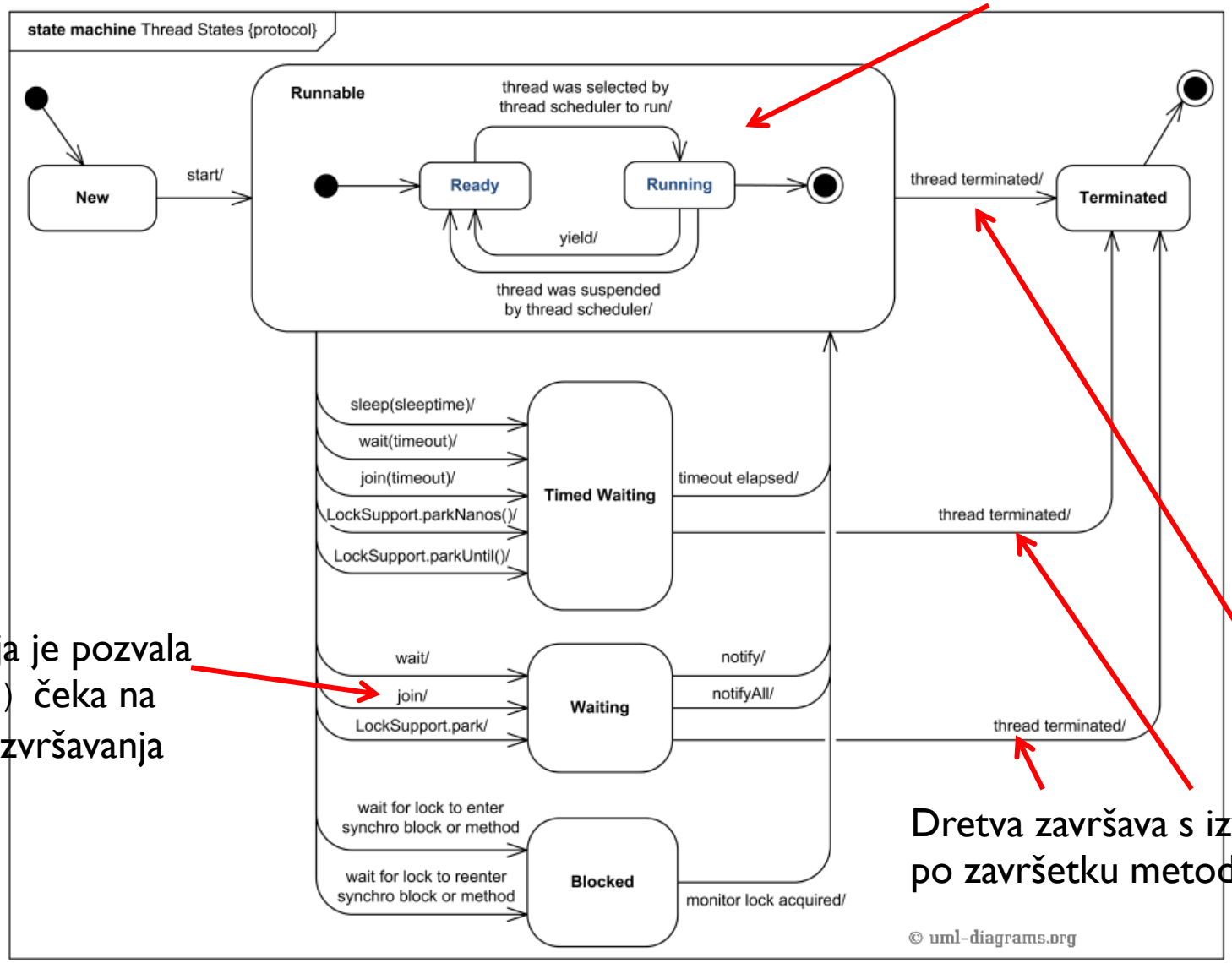
- ***Runtime data areas*** – memorijski prostori koje organizira JVM (ovisno od implementacije VM)
  - ***Method area*** – sadrži informacije o metodama, atributima, konstantama razreda i dr.
  - ***Heap*** – memorijski prostor za pohranjivanje objekata – tzv. **gomila**, rezerviran pri pokretanju VM, nad njim operira sakupljač smeća (*garbage collector*)
  - ***Java stacks*** i ***PC register*** – svaka dretva pri nastajanju dobije svoj stog i programsko brojilo (*PC register*). Stog je organiziran u okvire (*stack frames*), svaki okvir za jedan poziv metode – pamte se argumenti poziva, povratna vrijednost, lokalne varijable i međurezultati, samo je jedan okvir aktivan u jednom trenutku; programsko brojilo pokazuje adresu JVM instrukcije koja se izvodi
  - ***Native method stack*** – dretve koje izvršavaju nativne metode (metode ovisne o operacijskom sustavu), imaju za to rezerviran zasebni stog

Stanja dretvi u Javi

- **NEW** – dretva koja je instancirana (`Thread t = new Thread()`), ali nije još pokrenuta
- **RUNNABLE** – dretva koja je pokrenuta (`t.start()`), njezina metoda `run` se izvršava u JVM-u
- **BLOCKED** – dretva koja je blokirana čekajući na ulazak u kritični odsječak (monitor)
- **WAITING** – dretva čeka neodređeno dugo na drugu dretvu da izvrši neku akciju
- **TIMED_WAITING** – dretva čeka određeno vrijeme na drugu dretvu da izvrši neku akciju
- **TERMINATED** – dretva koja je završila izvođenje metode `run` se nalazi u ovom stanju

Stanja dretvi u Javi

Thread scheduler JVM-a upravlja redoslijedom izvođenja dretvi, pri čemu bitnu ulogu u odabiru redoslijeda izvođenja igra prioritet (*priority*) dretve



Dretva koja je pozvala `t.join()` čeka na završetak izvršavanja dretve `t`

Dretva završava s izvođenjem po završetku metode `run()`

Stanja dretvi u Javi

- Kritični odsječak u Javi može biti **metoda** ili **blok naredbi** ispred koje se nalazi ključna riječ `synchronized`, npr.

```
synchronized(obj) {
      while (<condition does not hold>) obj.wait();
      ...  // Perform action appropriate to condition
      notify();
}
```

- `synchronized` je Javin način za reći da postoji sinkronizacijski mehanizam – monitor za pristup kritičnom odsječku, samo jedna dretva mu može pristupiti u nekom trenutku
- Dretva koja poziva `wait()` ili `notify()` ili `notifyAll()` nad objektom mora se nalaziti u kritičnom odsječku (biti vlasnik monitora)
- Pozivom `wait()` dretva prestaje biti vlasnik monitora i čeka dok neka druga dretva ne pozove `notify()` nad tim objektom
- Pozivom `notify()` odabire se slučajno jedna dretva koja je čekala i koja se dalje natječe za izvođenje kritičnog odsječka kad trenutna dretva završi s njim
- Pozivom `notifyAll()` odabiru se sve dretve koje čekaju nad monitorom i onda se dalje natječu za izvođenje kritičnog odsječka

"Rješavanje" problema eksplozije broja stanja u JPF-u – smanjenje prostora stanja

- Prvi način: **provjera podudaranja stanja** (engl. *state matching*)
  - Na svakom grananju u kodu, provjerava se je li već istraženo isto stanje.
  - Ako se pronađe takvo stanje, onda se može sigurno obustaviti daljnja pretraga i vratiti se nazad na točku u kojoj još ima neistraženih putova
  - Dalje se od te ranije točke nastavlja pretraga prostora stanja.
  - Pri povratku na raniju točku, obnavlja se prijašnje stanje programa, što djeluje kao povratak debuggera unazad za *N* instrukcija!
  - Točno koje varijable programa ulaze u provjeru jednakosti stanja je nešto što se može korisnički odrediti

"Rješavanje" problema eksplozije broja stanja u JPF-u – smanjenje prostora stanja

- Drugi način: **djelomično smanjenje poretka** (engl. *partial order reduction*)
  - Broj različitih kombinacija pri raspoređivanju izvođenja dretvi je glavni razlog za veliki prostor stanja kod istovremenog izvođenja
  - Međutim, u većini praktičnih slučajeva nije potrebno razmatrati sve mogućnosti ispreplitanja izvođenja dretvi, budući da neki rasporedi izvršavanja programa dovode do istih stanja
  - Broj raspoređivanja dretvi smanjuje se tako da se grupiraju svi nizovi instrukcija u nekoj dretvi koji nemaju utjecaj izvan te dretve i to tako da se oni svedu na jedan prijelaz
  - Time se smanjuje do 70% prostora stanja programa
  - Djelomično smanjenje poretka ostvaruje se tako da se tijekom izvođenja prate instrukcije koje su bitne za raspoređivanje (npr. `synchronized` metode) ili one koje simuliraju nedeterminističku dodjelu vrijednosti varijabli te se prijelaz ostvaruje između takvih instrukcija.
  - Više na: https://github.com/javapathfinder/jpf-core/wiki/Partial-Order-Reduction

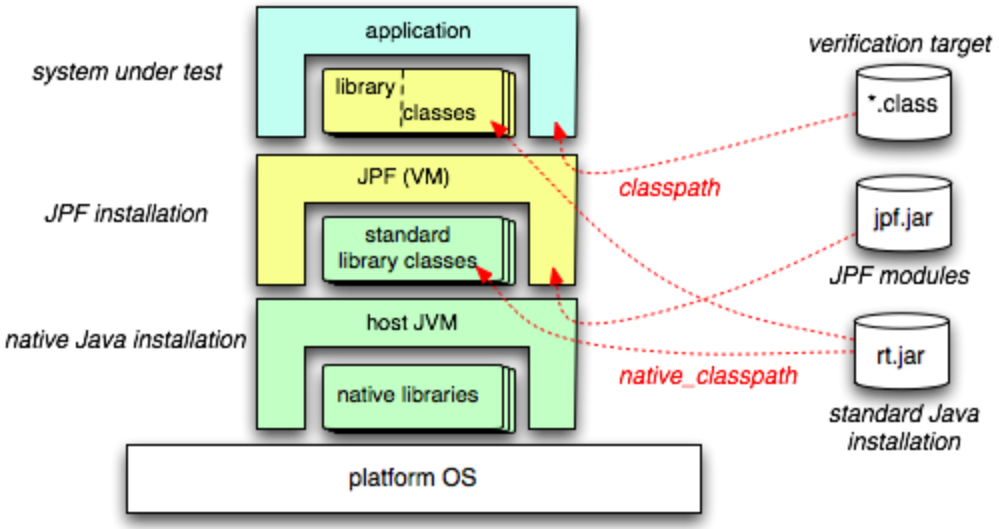
“Rješavanje” problema eksplozije broja stanja u JPF-u – smanjenje prostora stanja

- Treći način: **delegiranje izvođenja metoda JVM-u**
  - Neke instrukcije, i to one za koje se zna da ne utječu na verifikaciju (npr. `System.out.println()` delegiraju se za izvođenje host JVM-u umjesto da se izvršavaju unutar JPF-a

STRUKTURA JAVA PATHFINDERA

JPF i izvođenje programa

- JPF je ostvaren kao zasebni virtualni stroj (VM) koji se izvodi iznad instaliranog JVM-a za određeni OS (*host* JVM)
- JPF sam odlučuje koje dijelove našeg programa sam obrađuje, a koje delegira *host* JVM-u
- U pravilu, JPF značajno **usporava** izvođenje našeg programa (SUT) jer je to dodatni sloj iznad već postojećeg JVM-a



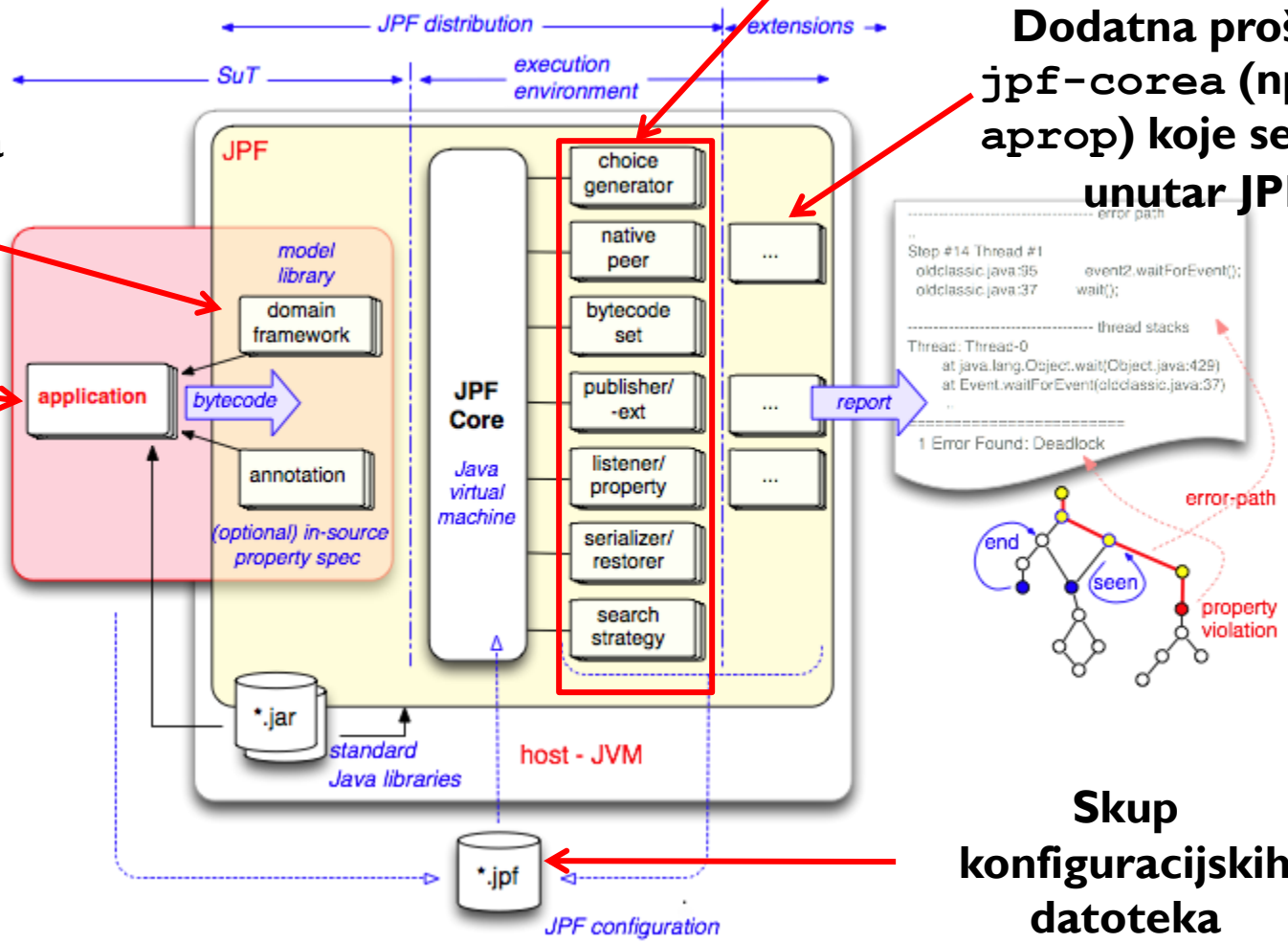
Struktura JPF-a

Standardni paketi `jpf-corea`

Dodatna proširenja `jpf-corea` (npr. `jpf-aprop`) koje se koriste unutar JPF-a

Knjižnice JPF-a koje koristi naša aplikacija

Naša aplikacija (datoteke *. class)



Skup konfiguracijskih datoteka

Struktura direktorija *src*

- **Main**
  - Paketi koji sadrže razrede ključne za rad `jpf-corea` (npr. algoritmi pretraživanja prostora stanja, izvršavanje VM, slušači i dr.). Oni se izvode na ***host* JVM-u** (kao što je specificirano u JPF `native_classpath`)
- **Peers**
  - *Native peer classes* – Paketi s ostalim razredima koji sadrže implementaciju metoda koje se izvode umjesto pravih nativnih metoda. Izvode se na ***host* JVM** (isto specificirano u JPF `native_classpath`)
- **Classes**
  - *Model classes* – Razredi koji se izvršavaju **izravno na VM-u od JPF-a**, a koje može importirati i koristiti naša aplikacija, služe kao knjižnice JPF-a za našu aplikaciju (specificirani su u JPF `classpath`)
- **Annotations**
  - Sadrži Javine anotacije (koje počinju sa znakom "@") koje treba obraditi JPF. Naša aplikacija ih može koristiti ako se provjeravaju neka svojstva (npr. @JPFConfig, @FilterField)
- **Examples**
  - Sadrži primjere SUT-ova i njima odgovarajućih konfiguracijskih datoteka sa specificiranim svojstvima
- **Tests**
  - Sadrži tipične ispitne slučajeve (testove) za veći broj standardnih Java razreda kao i za ispitivanje rada samog JPF-a

Struktura direktorija `build`

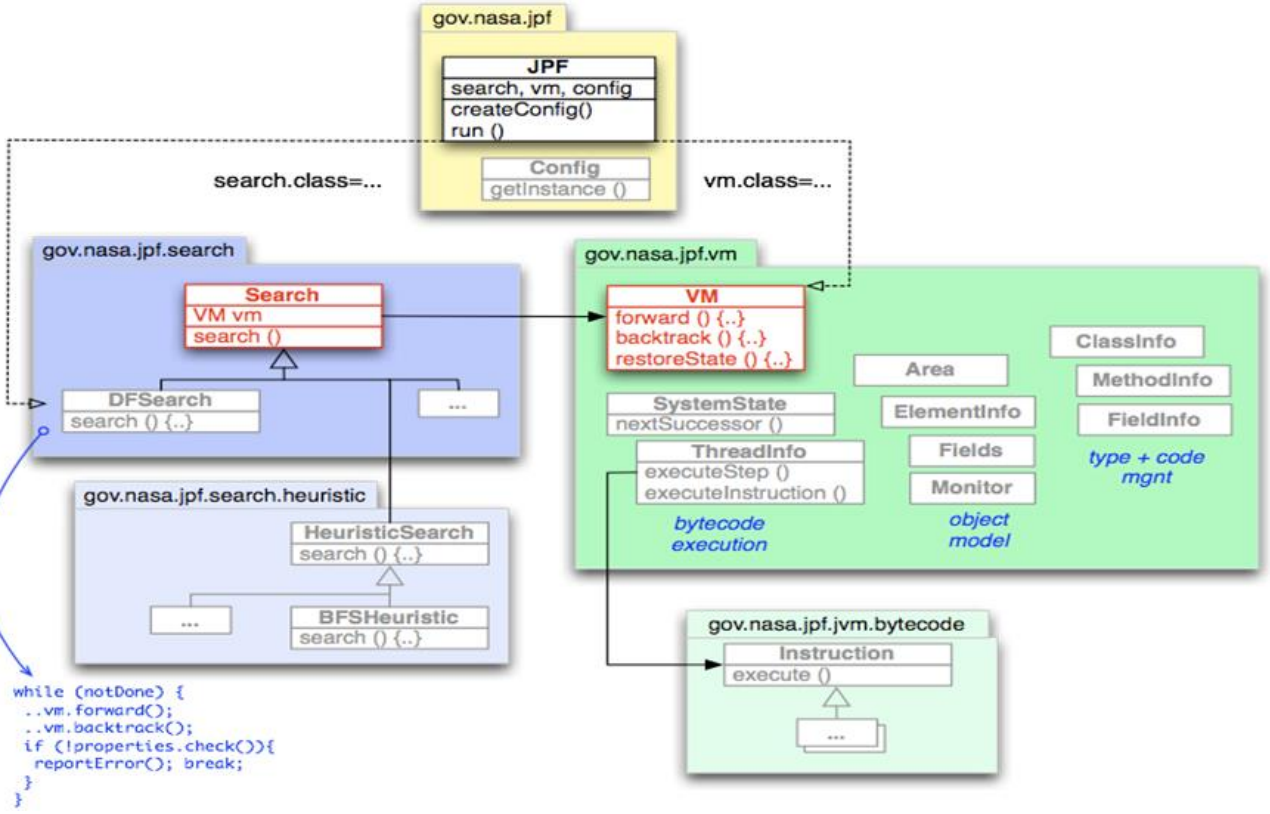
- Slična kao i struktura direktorija `src` (podjela na šest poddirektorija)
- Direktorij `build` stvara se izgradnjom (engl. *build*) – kompajliranjem svih razreda (`*.java`) iz direktorija `src`
- U tu svrhu koristi se sustav Ant za izgradnju aplikacije – postoji već unaprijed napisana skripta za izgradnju JPF-a koja se samo pokrene, najbolje iz razvojnog okruženja (npr. NetBeans) ili iz Anta
- Najznačajnija za pokretanje JPF-a je (Java-izvršna) datoteka `RunJPF.jar` koja je izgrađena u korijenu direktorija `build`

Ostali značajniji direktoriji

- Direktorij `lib`
  - Tu se stavljaju dodatne knjižnice koje se mogu koristiti uz `jpf-core` (npr. JUnit za ispitivanje)
- Direktorij `doc`
  - Dokumentacija `jpf-corea` u tekstualnom obliku (*.md – markdown *readme* datoteke)

Poddirektorij `main` – najvažniji dio `jpf-core`

- **VM** (`gov.nasa.jpf.vm.VM`) i **Search** (`gov.nasa.jpf.search.Search`) su najbitniji
- `VM` proizvodi stanja programa i pomiče se korak unaprijed ili unatrag
- `Search` je upravljački program nad `VM`-om koji omogućuje pretraživanje stanja programa korištenjem raznih strategija (DFSearch, RandomSearch…) i heuristika, ovisno o specifikaciji



SPECIFIKACIJA I PROVJERA SVOJSTAVA PROGRAMA

Specifikacija i provjera svojstava programa

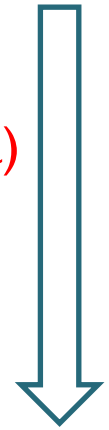
- Specifikacija svojstava koje treba provjeriti u JPF-u nije jednostavna kao u drugim sustavima (npr. NuSMV)
- Svojstva se ne navode u vremenskoj logici, već se navodi koja svojstva programa treba provjeriti (npr. Nonnull, Deadlock...)
- JPF je visoko konfigurabilan, čime je omogućena velika proširivost sustava, no nedostatak je velika složenost konfiguracije
- Različiti dijelovi sustava mogu imati različite vlastite parametre (vrsta pretrage, slušači, skupovi instrukcija...)
- Time je onemogućeno postojanje konfiguracijskog objekta koji bi sadržavao konkretna unaprijed definirana polja za parametre

Specifikacija i provjera svojstava programa

- Konfiguracijski objekt je fleksibilan:
  - zasnovan je na nizovima znakova (String)
  - po volji je proširiv
  - prenosi se odozgo prema dolje u hijerarhijskom procesu tako da svaka komponenta izvuče samo svoje potrebne parametre
- Konfiguracijski objekt ostvaren je razredom **`Config`** (u paketu `gov.nasa.jpf`)
- On obavještava daljnje razrede pri promjeni bilo kojih parametara, a daljnji razredi provjeravaju parametre i ako utvrde da su zaduženi za njihovu obradu, onda ih obrađuju

Specifikacija i provjera svojstava programa

- Konfiguracija sustava se ostvaruje na četiri razine:
  - Cjelokupne instalacije na računalu
  - Projektne instalacije (za svaku komponentu JPF-a koja je instalirana)
  - Korisničke aplikacije (SUT)
  - Komandno-linijski argumenti (prema potrebi)
- **Niža razina uvijek može nadjačati svojstva (konfiguraciju) više razine!**



Podešavanje cjelokupne instalacije

- Datoteka s parametrima za cjelokupnu JPF instalaciju naziva se **`site.properties`** i treba se nalaziti na lokaciji `<user.home>/.jpf/site.properties`
- To je uobičajena datoteka vrste ***Java properties,*** što znači da je većina svojstava definirana po principu:

  <ključ> = <vrijednost>
- Ova datoteka govori JPF-u prilikom pokretanja gdje da traži projekte u okviru JPF-a koji su instalirani na računalu, kako bi podesio njihove *classpathove* bez potrebe za time da ih korisnik sam svaki put mora iznova upisivati
- Datoteku `site.properties` potrebno je ručno napisati prilikom prve instalacije JPF-a, jer u suprotnom JPF neće raditi

Podešavanje cjelokupne instalacije

- Primjer datoteke `site.properties`:

```
# JPF site configuration

jpf.home = ${user.home}/Documents/NetBeansProjects

# can only expand system properties
jpf-core = ${jpf.home}/jpf-core
extensions=${jpf-core}

# annotation properties extension
jpf-aprop = ${jpf.home}/jpf-aprop
extensions+=,${jpf-aprop}

# numeric extension
jpf-numeric = ${jpf.home}/jpf-numeric
extensions+=,${jpf-numeric}
```

Definiramo lokaciju za `jpf.home`

Lokacija instalacije `jpf.core`

`extensions` definira sva proširenja (sve instalirane projekte). U novijim verzijama JPF-a `extensions` nije nužno eksplicitno navoditi u `site.properties`.

Lokacije ostalih projekata koje treba uzeti u obzir

Napomene: ostali projekti se mogu navesti u `site.properties` čak i ako nisu instalirani na računalu. JPF će ih u tom slučaju ignorirati. Dodatni projekt našeg SUT-a ne treba se navoditi u datoteci `site.properties`, ona služi samo za JPF!

Podešavanje projektne instalacije

- Svaki projekt u JPF-u (uključujući `jpf-core` i sva proširenja) sadržava u svojem **korijenskom** direktoriju datoteku `jpf.properties`
- U toj datoteci navode se svi parametri karakteristični za taj projekt u cjelini
- Tipično se tu navode *classpathovi* do svih JPF aplikacija koje projekt sadrži, svojstva koja se po *defaultu* provjeravaju kao i ostali parametri svojstveni tom projektu
- Svaki korisnički projekt koji će biti SUT također može (ali ne mora) u svojem korijenskom direktoriju imati datoteku `jpf.properties`
- Datoteke `jpf.properties` izvršavaju se onim redoslijedom koji je određen u datoteci `site.properties`, uz iznimku ako se JPF pokreće unutar određenog projekta, u kojem slučaju se najprije učitava `jpf.properties` za taj projekt
- Bitno je da su `jpf.properties` i `site.properties` međusobno konzistentne, što znači da nazivi projekata odgovaraju (npr. "`jpf-aprop`" mora biti isti u datotekama `site.properties` i `jpf.properties`)

Podešavanja projektne instalacije

- Primjer datoteke `jpf.properties`:

```
jpf-aprop = ${config_path}

#--- path specifications
jpf-aprop.native_classpath = build/jpf-aprop.jar;lib/antlr-runtime-3.1.3.jar
jpf-aprop.classpath = build/examples
jpf-aprop.test_classpath = build/tests
jpf-aprop.sourcepath = src/examples

#--- other project specific settings
listener=${listener.autoload},javax.annotation.Nonnull,...
listener.javax.annotation.Nonnull=gov.nasa.jpf.aprop.listener.NonnullChecker
```

Prvo svojstvo uvijek definira naziv projekta, {config_path} se uvijek proširuje nazivom direktorija gdje se nalazi `jpf.properties`

Put do razreda koji će izvršavati programe, a koji mora biti vidljiv *host* JVM-u

Put do direktorija koji sadrže primjere (SUT)

Put do direktorija koji sadrže uobičajene testove

Put do direktorija koji sadrže izvorni kod, što se koristi ako JPF treba generirati trag programa

Postavljanje dodatnih svojstava specifičnih za projekt `jpf-aprop`, konkretno u ovom slučaju učitava se slušač (*listener*) koji provjerava anotacijsko svojstvo @Nonnull. Sada više nije potrebno u SUT konfiguracijskim datotekama navoditi tog slušača!

Podešavanje svojstava SUT-a

- Put do direktorija gdje se nalaze datoteke `*.class` SUT-a definira se najčešće u datoteci `jpf.properties` (svojstvo `.classpath`)
- Podešavanje svojstava **specifičnih za određeni SUT** navodi se u datoteci `*.jpf`, koja se može nalaziti bilo gdje, no obično u istom direktoriju gdje je i izvorni kod samog razreda koji se provjerava
- Uobičajeno je dati naziv toj datoteci takav da odgovara nazivu razreda aplikacije koja se ispituje, npr. `oldclassic.jpf`, ali to nije nužan uvjet
- Kako bi se pokrenuo JPF, u datoteci `*.jpf` **nužno** je definirati **koji glavni razred aplikacije (razred koji sadrži metodu `main`) je potrebno izvršiti** (ključ `target`)
- Mogu se navesti argumenti koje treba primiti `static` metoda `main` tog glavnog razreda (ključ `target_args`)
- Osim toga, navodi se niz svojstava koja specificiraju kako želimo provjeriti našu aplikaciju (slušači, način pretraživanja, način i poredak izvještavanja...)

Podešavanje svojstava SUT-a

- Primjer datoteke (skraćeno) `RobotManager.jpf`

```
#--- dependencies on other JPF modules
@using = jpf-awt
@using = jpf-shell

#--- what JPF should run
target = RobotManager
```

"@using=<ime_projekta>" je naredba JPF-u da učita `jpf.properties` datoteku navedenog projekta (koji mora biti definiran u datoteci `site.properties`). Ovako se uklanja potreba za navođenjem `extensions` u datoteci `site.properties`

Definiranje naziva ciljnog glavnog razreda aplikacije koji se pokreće da bi izvršila provjera modela

Dodatni slušač, uz one navedene u `jpf.properties` datotekama koje se uzima u obzir. Ovaj konkretni slušač nalazi se u `jpf-core: gov.nasa.jpf.listener.OverlappingMethodAnalyzer`

```
#--- other stuff that defines how to run JPF
listener+=,.listener.OverlappingMethodAnalyzer
cg.enumerate_random=true
```

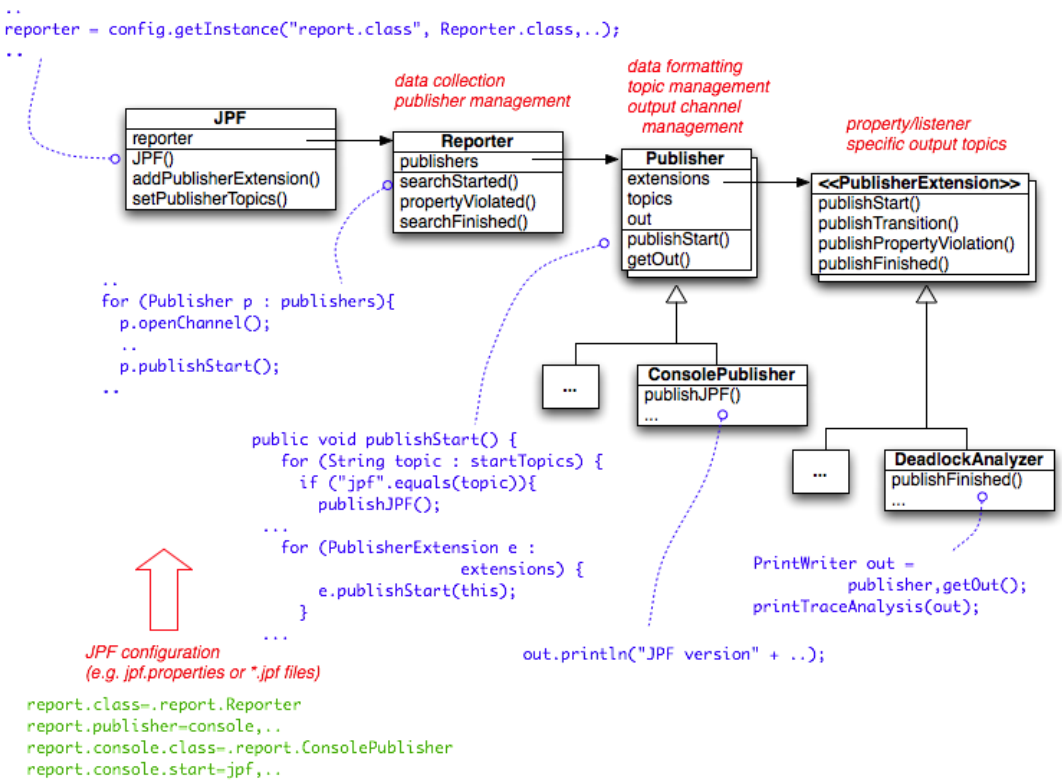
Specifikacija svojstva da se pobroje sve mogućnosti prilikom slučajnog izbora koji se pojavljuje unutar programa (npr. `Random.nextInt(2);` će dati dvije mogućnosti: 0 ili 1). Ključ `cg.enumerate_random` ispituje se *u Native peeru* `JPF_java_util_Random` projekta `jpf-core`

Podešavanje svojstava korištenjem komandne linije

- Komandnom linijom JPF-a moguće je nadjačati sve gornje razine hijerarhije konfiguracije
- U tom slučaju, najčešće se dodaju svojstva korištenjem notacije `<ključ>+=<vrijednost>`, iako su moguće i drugačije notacije, vidjeti:
- https://github.com/javapathfinder/jpf-core/wiki/Configuring-JPF (dio *Special Property syntax*)
- Najfleksibilnije, ali uglavnom nepotrebno rješenje osim u slučaju *debuggiranja* samog JPF-a
- Mora se točno znati što se radi ☺

Izvještaji

- Za generiranje konačnog izvještaja (engl. *report*) o provjeri modela zaduženi su:
  - Razred izvjestitelj - Reporter
  - Razredi izdavači - Publisher, čiji tipovi ovise o specifikaciji vrste izvještavanja
  - Razredi koji proširuju izdavače - PublisherExtensions – specifični izdavači za određena svojstva



Izvještaji

- Izvjestitelj upravlja i obavještava izdavače kada se dođe do pojedine faze izvještavanja (*start, property_violation, finish*)
- Faza *property_violation* ima određene teme izvještavanja
  - ***error*** – prikazuje tip i detalje kršenja svojstva koje je pronađeno
  - ***trace*** – prikazuje trag programa koji dovodi do kršenja svojstva
  - ***snapshot*** – daje listu stanja svake dretve u trenutku kršenja svojstva
  - ***output*** – prikazuje izlaz iz programa za taj trag
- Faza *finish* ima po *defaultu* postavljene teme izvještavanja
  - ***results*** – prikazuje je li došlo do kršenja svojstava i daje kratku listu pogrešaka
  - ***statistics*** – prikazuje ukupnu statistiku izvođenja JPF-a na SUT-u
- Izdavači proizvode izlaz iz sustava u željenom obliku (npr. tekst, XML). Uobičajeni izdavač je `ConsolePublisher` (`report.console.*` za izvještavanje u konzoli u obliku teksta)
- Za svaki simbolički naziv izdavača i fazu izvještavanja potrebno je specificirati kojim redom se teme prikazuju (u datoteci `*.jpf`):

Primjer:

```
report.console.property_violation=error,trace,snapshot
```

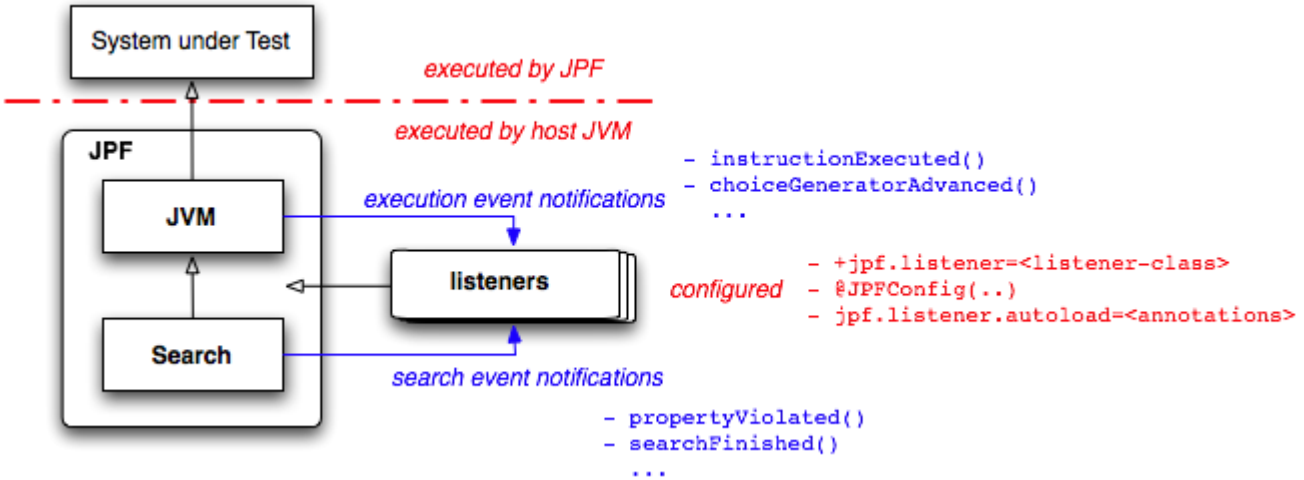
NAPREDNE TEME I PROŠIRENJA

Napredne teme i proširenja JPF-a

- Slušači (engl. *Listeners*)
- Generatori izbora (engl. *Choice generators*)
- Provjera anotacijskih svojstvava – `jpf-aprop`

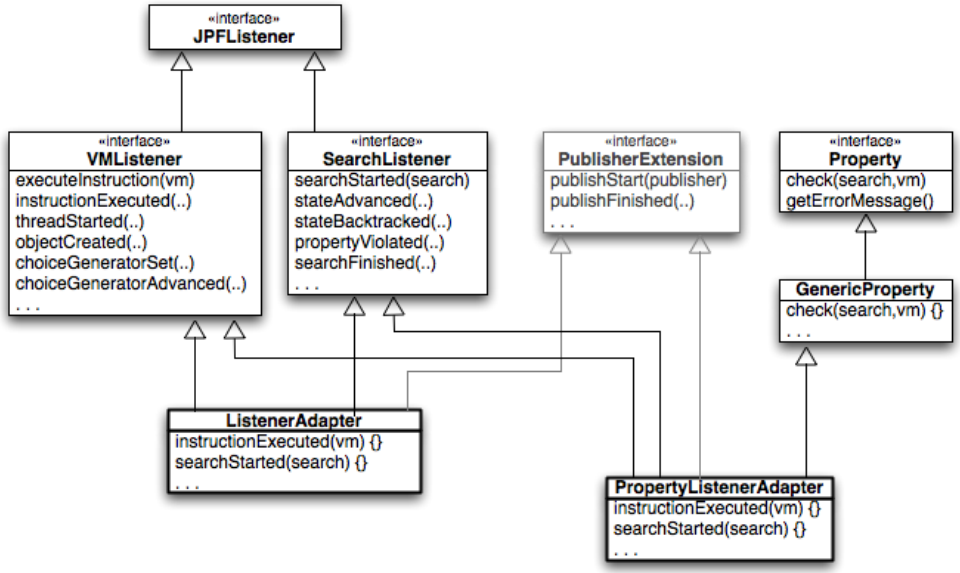
Slušači

- Najvažniji mehanizam proširenja u JPF-u
- Pružaju način za opažanjem, interakcijom i proširenjem izvođenja JPF-a
- Slušači su *Observeri* koji reagiraju na određeni događaj prilikom pretrage (`SearchListener`) ili rada JVM-a od JPF-a (`VMListener`)
- Konfiguriraju se navođenjem svojstva `listener` u `jpf.properties` ako vrijede za cijeli projekt, u datoteci `*.jpf` (najčešće), ili kroz komandnu liniju
- JPF također pokreće odgovarajućeg pridruženog slušača kad naiđe na anotaciju @JPFConfig s navedenim slušačem u datoteci s izvornim kodom



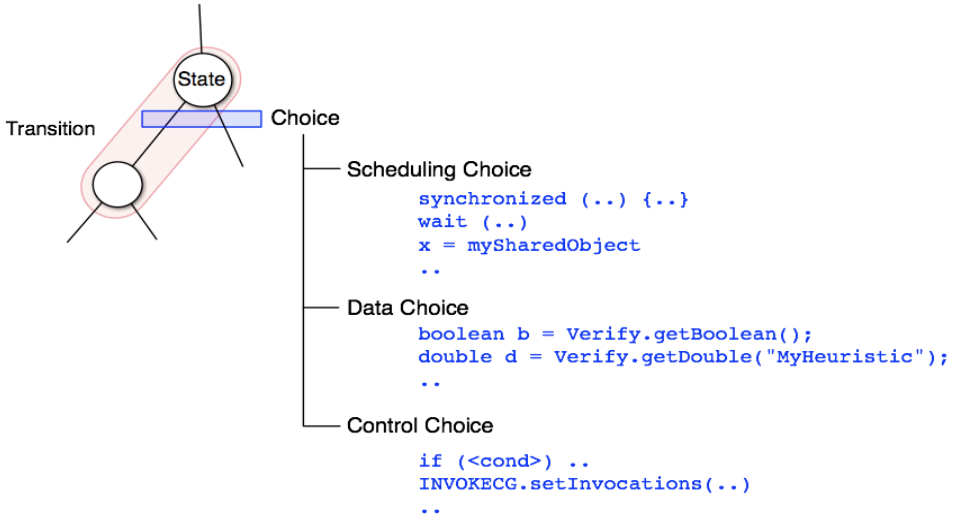
Slušači

- Konkretni slušači obično nasljeđuju jedan razred *Adapter* koji ima prazne implementacije odgovarajućih metoda sučelja `VMListener`, `SearchListener` i drugih (npr. `NullTracker` **`extends`** `ListenerAdapter`) i zatim nadjačavaju metode dojave za koje su zainteresirani
- `ListenerAdapter` služi za skupljanje informacija o izvođenju JPF-a pa će ga koristiti slušači npr. `CoverageAnalyzer`, `DeadlockAnalyzer` i `NullTracker`
- `PropertyListenerAdapter` se koristi kada slušač implementira neko svojstvo programa povezano s pretragom (npr. `PreciseRaceDetector`, `NoStateCycles`)



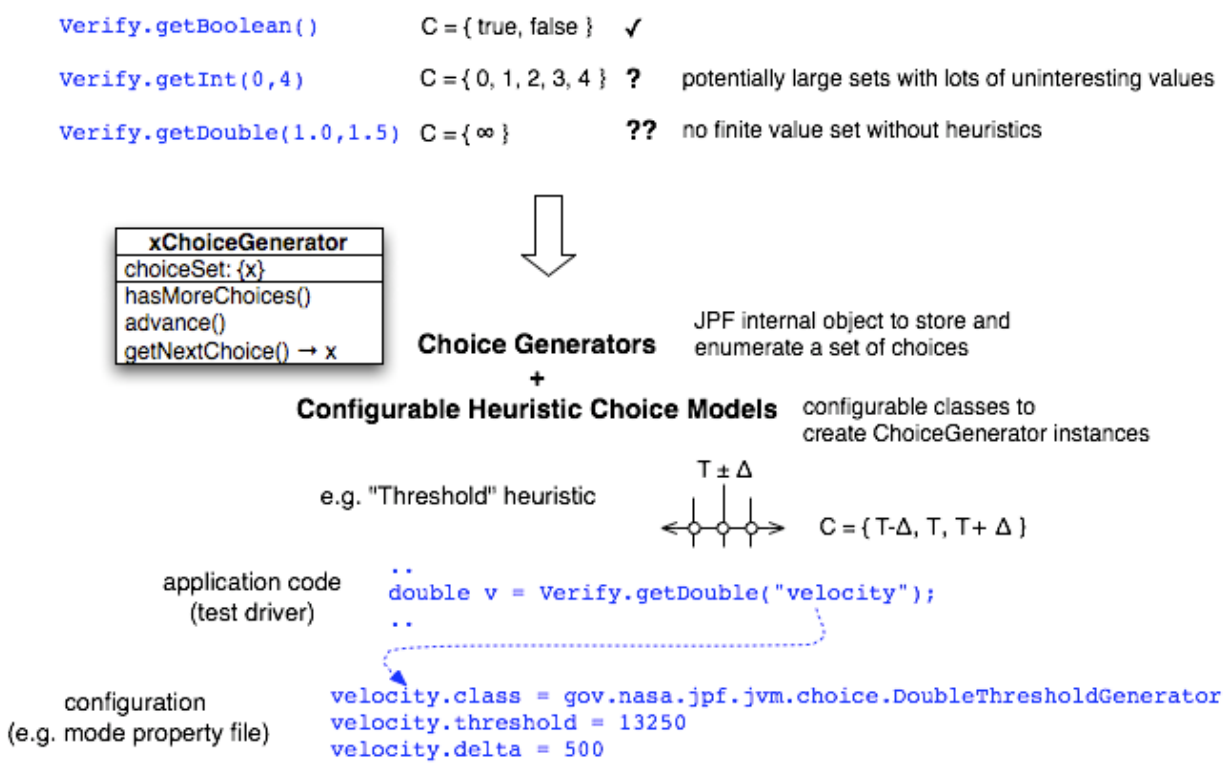
Generatori izbora

- Provjera modela ima zadatak doći do zanimljivih stanja programa uz ograničene resurse koji su na raspolaganju
- *ChoiceGenerators* je mehanizam JPF-a da sustavno istražuje prostor stanja kako bi došao do rješenja
- Veći broj postojećih mogućnosti izbora: **raspoređivanje dretvi, vrijednosti podataka, kontrolni tok programa**
- Mehanizam odabira odgovarajućeg generatora izbora i parametara heuristike odvojen je od samog programa – navodi se u konfiguracijskoj datoteci kao svojstvo odgovarajućeg imena zajedno s ostalim svojstvima



Generatori izbora

- Kod provjere vrijednosti podataka, kako bi se smanjio broj mogućnosti dodjele vrijednosti pri provjeri `int`, `double` i drugih tipova varijabli, uvode se heuristike pretraživanja



Generatori izbora

- Oprez: korištenjem heuristika odstupa se od temeljnog (ali idealističkog) zahtjeva kod provjere modela: da se ispitaju svi putovi kroz program, u ovom slučaju ispitaju se samo oni zanimljivi putovi (zanimljivost je subjektivan kriterij!)
- Interno, u JPF-u, generatori izbora omogućuju odabir pri postupku raspoređivanja dretvi našeg programa koje se u JPF-u izvodi kao *bytecode* (`ChoiceGenerator` i `ThreadChoiceGenerator`),
- Detaljnije: https://github.com/javapathfinder/jpf-core/wiki/ChoiceGenerators

Provjera anotacijskih svojstava

- Projekt `jpf-aprop` predstavlja proširenje `jpf-corea` s namjenom provjere modela za specificirana anotacijska svojstva programa
- Ako se u nekom trenutku žele provjeriti anotacijska svojstva, odgovarajući slušači će se navesti u `*.jpf` konfiguracijskoj datoteci te će se program moći prevesti i provjeriti
- U idealnom slučaju, ove anotacije su korisne i za dokumentiranje programa, a može ih se obraditi i s nekim alatima za statičku analizu koda
- Izvedbeno, prije prevođenja SUT-a, potrebno je napraviti *import* odgovarajućih razreda koji su zaduženi za dotične anotacije te uključiti knjižnicu `jpf-aprop-annotations.jar` u kojoj se nalazi popis razreda s anotacijama.

Provjera anotacijskih svojstava

- Struktura projekta `jpf-aprop` je slična projektu `jpf-core`, samo je `jpf-aprop` bitno manji i ima jednostavniju datoteku `jpf.properties`
- Anotacijska svojstva pokrivaju sljedeće zadatke:
  - Nedozvoljeno dodjeljivanje `null` vrijednosti (@Nonnull)
  - Ugovore (@Requires, @Ensures, @Invariant)
  - Obilježja sigurnosti pristupa polju razreda kod dretvi (@GuardedBy)
  - Promijenjivost objekta (@Const)
  - i drugo...

INSTALACIJA

Instalacija JPF-a

1. Instalacija Jave i NetBeans IDE-a
2. Kloniranje repozitorija (`jpf-core`) s GitHuba
3. Izgradnja projekta `jpf-core` korištenjem Ant-skripte `build.xml`
4. Izrada datoteke `site.properties`
5. Instalacija *plugina* za NetBeans za provjeru modela ("Verify...")
6. Spremni! ☺

O 2. domaćoj zadaći

- Instalacija JPF-a

- Sama zadaća je podijeljena u 4 dijela:

1. Upoznavanje s projektom `jpf-core` i pokretanje provjere modela jednostavnih primjera programa.
2. Uvode se dodatni razredi slušači koji nadograđuju osnovnu funkcionalnost projekta `jpf-core`.
3. Pokriveno je izvođenje provjere modela nad primjerima iz dodatnog projekta `jpf-aprop`
4. Verifikacija korisničkog projekta, uključivanje korištenja projekata `jpf-core` i `jpf-aprop` te provjera modela zadanog programa uz stalne izmjene.

O čemu nismo detaljno govorili

- Model Java Interface (MJI)
  - Mehanizam preusmjeravanja izvođenja metoda na JPF-u ili od JPF-a na JVM
  - https://github.com/javapathfinder/jpf-core/wiki/Model-Java-Interface
- *Bytecode factories*
  - Kako JPF interno procesira instrukcije *bytecodea* programa
  - https://github.com/javapathfinder/jpf-core/wiki/Bytecode-Factories
- *Logging*
  - Bilježenje pogrešaka u ovisnosti o ozbiljnosti (*severity*) pogreške u dnevnik (*log*)
  - https://github.com/javapathfinder/jpf-core/wiki/Logging-system
- *Symbolic Pathfinder*
  - Izvršavanje programa korištenjem simboličkih (ograničenih numeričkih) vrijednosti ulaznih varijabli na temelju analize koda – koristi se uglavnom za automatizaciju procesa ispitivanja
  - https://github.com/SymbolicPathFinder/jpf-symbc/wiki
- Velik broj ostalih proširenja:
  - https://github.com/javapathfinder/jpf-core/wiki/Projects

Zaključak

- JPF je napredan i proširiv radni okvir za provjeru modela programa pisanog u Javi
- Nadograđuje Javin VM svojim vlastitim VM što mu omogućuje nesmetano kretanje kroz stanja programa
- Koristi više tehnika za smanjenje prostora pretraživanja
- Relativno složena konfiguracija projekata koja omogućava proširivost, ali i komplicira uporabu